ความสัมพันธ์เวียนเกิด (recurrence relation) และฟังก์ชันเรียกตัวเอง (recursive function)

1. บทนำ

เอกสารชุดนี้มีวัตถุประสงค์เพื่อนำเสนอวิธีการเขียนโปรแกรมโดยอาศัยฟังก์ชันแบบเรียกตัวเอง (recursive function) โดยเนื้อหาประกอบด้วย ความรู้เกี่ยวกับความสัมพันธ์เวียนเกิด (recurrence relation) การวิเคราะห์ปัญหา เพื่อหาความสัมพันธ์ของลำดับตัวเลขหรือนิพจน์ ซึ่งเป็นจุดเริ่มต้นของการนำไปสู่การออกแบบ และเขียนฟังก์ชันเรียกตัวเอง ที่สามารถช่วยให้การเขียนโปรแกรมง่ายและกระชับขึ้น

2. ความสัมพันธ์เวียนเกิด (recurrence relation)

ปัญหาโดยทั่วไปเกี่ยวกับความสัมพันธ์เวียนเกิด คือ การวิเคราะห์หาความสัมพันธ์ของลำดับตัวเลข เช่น

1 2 3 4 5 6 ?

1 1 2 3 5 8 ?

กล่าวคือ พยายามมองหาความสัมพันธ์ของลำดับตัวเลขระหว่างพจน์ถัดไปกับพจน์ในลำดับก่อนหน้า (ซึ่งอาจมีได้มากกว่า 1 พจน์)

ความหมาย

ความสัมพันธ์เวียนเกิดสำหรับลำดับ a_n คือ สมการที่แสดงความสัมพันธ์ระหว่างพจน์ a_n กับพจน์ก่อนหน้า ซึ่งอาจเป็น a_{n-1}, a_{n-2}, ... a_1, a_0 (อย่างน้อย 1 พจน์) เมื่อ $n \geq n_0$ และ n_0 เป็นจำนวนเต็มที่มากกว่าหรือเท่ากับ 0

ตัวอย่างเช่น

$a_n = a_{n-1} + 1$ *กรณีความสัมพันธ์ที่เกิดจากพจน์ก่อนหน้าเพียงหนึ่งพจน์

$a_n = a_{n-1} + a_{n-2}$ *กรณีความสัมพันธ์ที่เกิดจากพจน์ก่อนหน้ามากกว่าหนึ่งพจน์

ตัวอย่างที่ 2.1 ในห้องทดลองทางชีววิทยาแห่งหนึ่งพบว่า แบคทีเรียจะมีจำนวนเพิ่มขึ้นเป็นสองเท่าในทุก ๆ หนึ่งชั่วโมง สมมติเริ่มต้นมีจำนวนแบคทีเรียอยู่ 5 ตัว จงหาว่าจำนวนแบคทีเรียทั้งหมดเมื่อเวลาผ่านไป n ชั่วโมง

เมื่อพิจารณาถึงการเพิ่มจำนวน พบว่าลำดับของจำนวนจะเป็น 5 10 20 40 ...
จะเห็นว่าลำดับถัดไปจะมีค่าเป็นสองเท่าของลำดับก่อนหน้านั้น
ดังนั้น ความสัมพันธ์เวียนเกิดสำหรับปัญหานี้ คือ $a_n = 2a_{n-1}$
ผลการกระจาย เพื่อนำไปสู่การหาคำตอบเฉพาะ

$a_n = 2a_{n-1}$
$= 2 \times 2a_{n-2} = 2^2a_{n-2}$
$= 2 \times 2 \times 2a_{n-3} = 2^3a_{n-3}$
...
$a_n = 2 \times 2 \times 2 \ldots 2a_0 = 2^na_{n-n}$
$= 2^na_0$

- เงื่อนไขเริ่มต้น คือ $a_0 = 5$
- คำตอบเฉพาะ คือ $a_n = 2^n \times 5$
  ซึ่ง เป็นสูตรทั่วไปที่สามารถคำนวณหาค่าพจน์ใด ๆ ของลำดับได้ ส่วนวิธีการคำนวณหาคำตอบเฉพาะจะไม่ขออธิบายในที่นี้ หากต้องการศึกษาเพิ่มเติมสามารถค้นหนังสืออ่านเพิ่มเติมได้ในหัวข้อ การหาคำตอบเฉพาะของความสัมพันธ์เวียนเกิดเชิงเส้น

ตัวอย่างที่ 2.2 ถ้าลำดับ $\{a_n\}$ สอดคล้องกับความสัมพันธ์เวียนเกิด $a_n = na_{n-1}$ สมมติ $a_1 = 1$ จงหาว่าพจน์ที่ 3 และ 4 (a_3 และ a_4) มีค่าเท่าใด

ตัวอย่างที่ 2.3 ถ้าลำดับ $\{a_n\}$ สอดคล้องกับความสัมพันธ์เวียนเกิด $a_n = a_{n-1} + a_{n-2} + a_{n-3}$ เมื่อ $a_0 = a_1 = 1$ และ $a_2 = 2$ แล้ว จงหาว่าพจน์ที่ 4 และ 5 (a_4 และ a_5) มีค่าเท่าใด

ตัวอย่างที่ 2.4 กบตัวหนึ่งต้องการกระโดดขึ้นบันไดสูง n ขั้น ในการกระโดดแต่ละครั้งกบตัวนี้สามารถกระโดดได้ 1 ขั้น หรือ 2 ขั้น จงหาความสัมพันธ์เวียนเกิดสำหรับ a_n เมื่อ a_n คือจำนวนวิธีในการกระโดดขึ้นบันได n ขั้นของกบตัวนี้

ตัวอย่างที่ 2.5 จงหาความสัมพันธ์เวียนเกิดของการทำแต้มในกีฬาบาสเก็ตบอล ซึ่งสามารถทำแต้มได้ครั้ง 1 หรือ 2 หรือ 3 แต้มอย่างใดอย่างหนึ่งในแต่ละครั้งที่ชูทลง จนกระทั่งได้แต้มรวมทั้งสิ้น n แต้ม (สมมติ a_n คือจำนวนวิธีของการทำแต้ม n แต้ม)

ตัวอย่างที่ 2.6 จงหาความสัมพันธ์เวียนเกิดของการบริจาคเงิน 5 บาท หรือ 10 บาท อย่างใดอย่างหนึ่งในแต่ละวัน จนกระทั่งได้เงินบริจาครวมทั้งสิ้น n บาท (สมมติ a_n คือจำนวนวิธีของการบริจาคเงิน n บาท โดยที่ n หาร 5 ลงตัว)

3. ฟังก์ชัน (function)

ฟังก์ชัน คือ ส่วนของการทำงานย่อยในโปรแกรม ซึ่งมีวัตถุประสงค์ของการทำงานอย่างใดอย่างหนึ่งชัดเจน ส่วนย่อยสามารถเรียกใช้งานซ้ำ ๆ กี่รอบก็ได้ขึ้นอยู่กับความต้องการของโปรแกรม

หากเปรียบโปรแกรมเป็นเสมือนบ้านหลังหนึ่ง ฟังก์ชันย่อยในโปรแกรมก็เปรียบดั่งห้องต่าง ๆ ภายในบ้านที่ถูกแบ่งให้เป็นไปตามวัตถุประสงค์การใช้งานของห้องนั้น ๆ เช่น ห้องครัว ห้องน้ำ และห้องนอน บ้านก็ดูเป็นระบบระเบียบขึ้น แต่ละห้องสามารถถูกใช้ซ้ำหลาย ๆ รอบได้โดยไม่ต้องสร้างใหม่ บางห้องใช้ร่วมกันได้ การดูแลรักษาก็ทำได้ง่ายขึ้น (ลองมองเปรียบเทียบกับบ้านที่มีเพียงห้องเดียว แล้วรวมทุกอย่างไว้ในห้องนั้น ไม่ว่าจะเป็นการนอน ทำกับข้าว หรือแม้แต่อาบน้ำ เสมือนโปรแกรมที่มีเพียงฟังก์ชัน main เพียงห้องเดียว)

การเขียนโปรแกรม ก็เช่นกัน หากแบ่งโปรแกรมออกเป็นหน่วยย่อยตามฟังก์ชันการใช้งาน แทนที่จะรวมทุกอย่างไว้ในฟังก์ชัน main จะช่วยให้โปรแกรมมีความเป็นระบบมากขึ้น ช่วยลดความซ้ำซ้อนของคำสั่ง และช่วยให้การตรวจสอบแก้ไขโปรแกรมง่ายขึ้น

โครงสร้างของฟังก์ชัน

```
return_type  function_name (list of parameters) {
      declaration of local variables;
      statements;
      return (value); //สอดคล้องกับ return_type
}
```

เมื่อ `function_name` คือ ชื่อของฟังก์ชันที่ต้องเป็นไปตามหลักการตั้งชื่อ

`parameters` คือ รายการพารามิเตอร์สำหรับรอรับค่าที่ถูกส่งมายังฟังก์ชัน ซึ่งสามารถมีได้มากกว่า 1 พารามิเตอร์ โดยใช้เครื่องหมายจุลภาค (,) คั่นแต่ละพารามิเตอร์

`return_type` คือ ชนิดข้อมูลที่จะถูกส่งกลับ ซึ่งต้องสอดคล้องกับชนิดของค่า (`value`) ที่จะถูกส่งกลับผ่านคำสั่ง `return` แต่หากเป็นฟังก์ชันนั้นไม่มีการส่งค่ากลับ `return_type` จะถูกกำหนดเป็น `void`

`local variables` คือ รายการตัวแปรที่ประกาศใช้เฉพาะในฟังก์ชัน

ตัวอย่างที่ 3.1 ฟังก์ชันสำหรับคำนวณหาค่าสัมบูรณ์ และการใช้งานผ่านฟังก์ชัน main

```
#include <stdio.h>
int getAbsolute(int x) { //ฟังก์ชันสำหรับคำนวณหาค่าสัมบูรณ์ของพารามิเตอร์ x
      if(x<0)
            return -x;
      else return x;
}
int main() {
      int num, abs;
      scanf("%d", &num);
      abs = getAbsolute(num);
      printf("The absolute value = %d", abs);
 }
```

ตัวอย่างที่ 3.2 โปรแกรมสำหรับแสดงลำดับตัวเลขจำนวนเต็มสองจำนวน โดยให้จำนวนที่น้อยกว่าอยู่ทางซ้าย

```
#include <stdio.h>
void showOrdered(int n1, int n2);
int main() {
      int num1, num2;
      printf("Enter number 1 and number 2 : ");
      scanf("%d %d", &num1, &num2);
      showOrdered(num1, num2);
}
void showOrdered(int n1, int n2) {
      if(n1 < n2) {
            printf("%d %d", n1, n2);
      else  printf("%d %d", n2, n1);
}
```

4. ฟังก์ชันเรียกตัวเอง (recursive function)

ฟังก์ชันเรียกตัวเอง คือ รูปแบบของฟังก์ชันที่มีการเรียกใช้ตัวมันเองซ้ำ ๆ ไปจนกระทั่งเจอเงื่อนไขที่ทำให้หยุด โดยรูปแบบของการเรียกตัวเอง (recursion) ประกอบด้วย 2 ส่วนสำคัญ คือ

1) Recursive form
2) Stop condition

Recursive form คือ รูปแบบของการเรียกใช้ตัวเอง เป็นรูปแบบของการเขียนปัญหาให้อยู่ใน รูปสมการแสดงความสัมพันธ์ระหว่างพจน์ที่ n กับพจน์ในลำดับก่อนหน้า

เช่น $a_n = 2a_{n-1}$ หรือ $a_n = a_{n-1} + a_{n-2}$ เป็นต้น

Stop condition คือ เงื่อนไขเพื่อหยุดการเรียกใช้ตัวมันเอง ซึ่งจะมีความสอดคล้องกับ พจน์เริ่มต้นของ recursive form นั้น

จากความสัมพันธ์เวียนเกิดและปัญหาเกี่ยวกับความสัมพันธ์เวียนเกิด เราสามารถนำเอาวิธีการเขียนโปรแกรมที่อาศัย recursive function มาช่วยในการแก้ปัญหาดังกล่าวได้ เช่น การหาแฟกตอเรียล (factorial) การหาค่าไฟโบนาสกี (Fibonacci) ปัญหาหอคอยฮาร์นอย (tower of Hanoi) และปัญหา L-Shape เป็นต้น

ตัวอย่างที่ 4.1 การหาผลบวกของเลขจำนวนเต็มตั้งแต่ 1 ถึง n มีความสัมพันธ์ดังนี้

Recursive form คือ `sum(n) = n + sum(n-1)` เมื่อ `n > 1`

เงื่อนไขหยุด (stop condition) คือ `n==1` ซึ่งให้ค่า `sum(n) = sum(1) = 1`

```
int sum(int n) {       //เมื่อ n>=1 (pre-condition, assumption)
      if (n==1)        //กรณี n = 1 (stop condition)
            return(n);
      else             //กรณี n > 1
            return(n + sum(n-1));      //recursive call
}
```

ตัวอย่างการเรียกใช้งานฟังก์ชัน `sum(n)`

```
#include <stdio.h>
int sum(int n);                          /*function signature*/
int main() {
      int n;
      printf ("n = ");
      scanf("%d",&n);
      printf ("sum 1..n = %d", sum(n)); //start calling sum(n)
}
int sum(int n) {       //เมื่อ n >= 1
      if (n==1)        //กรณี n = 1
            return(n);
      else             //กรณี n > 1
            return(n + sum(n-1));      /*recursive call*/
}
```

ตัวอย่างที่ 4.2 การคำนวณหาค่า factorial ด้วยฟังก์ชันเรียกตัวเอง

n! = n * (n-1)* (n-2) * ... * (1) หรือ n! = n * (n-1)! เมื่อ 0! = 1! = 1 และ n >= 0

เขียนอยู่ในรูป recursive form ได้เป็น `factorial(n) = n*factorial(n-1)`
เมื่อ `factorial(n)` แทน n!

เงื่อนไขหยุด (stop condition) คือ `n==0||n==1` ซึ่งให้ค่า `factorial(n) = 1`

```
#include <stdio.h>
long int factorial (int n);              /*function signature*/
int main() {
      int n;
      printf("n  = ");
      scanf("%d",&n);
      printf("n! = %ld", factorial(n));
}
long int factorial(int n) {
      if(n==0 || n==1)                   /*stop condition*/
            return (1);
      else
            return (n*factorial(n-1)); /* recursive form */
}
```

ตัวอย่างที่ 4.3 แสดงการแปลงความสัมพันธ์เวียนเกิดของการเพิ่มจำนวนของแบคทีเรียตัวอย่างที่ 2.1 (ในหัวข้อที่ 2 ความสัมพันธ์เวียนเกิด) เป็นฟังก์ชันเรียกตัวเอง

รูปแบบของการเรียกตัวเอง คือ $a_n = 2a_{n-1}$ เมื่อ a_n คือ จำนวนแบคทีเรีย ณ ชั่วโมงที่ n

เงื่อนไขหยุด คือ n==0 ซึ่งให้ค่า $a_0 = 5$ (เป็นพจน์เริ่มต้นของความสัมพันธ์เวียนเกิด ที่จะกลายมาเป็นเงื่อนไขจบของฟังก์ชันเรียกตัวเอง)

สมมติแทน a_n ด้วยฟังก์ชัน `numbac(n)` สามารถเขียนฟังก์ชันเรียกตัวเองได้ดังนี้

```
#include <stdio.h>
int numbac(int n) {   //number of bacteria at nth hour
      if (n==0)                          /*stop condition*/
            return(5);
      else
            return (2*numbac(n-1));     /* recursive form */
}
int main() {
      int n;
      printf ("Enter the number of hours?");
      scanf("%d", &n);
      printf ("After %d hours = %d", n, numbac(n));
}
```

ตัวอย่างที่ 4.4 โปรแกรมคำนวณหาเลขยกกำลัง x^n เมื่อ n มีค่ามากกว่าหรือเท่ากับ 0

สมมติให้ `pow(x,n)` แทนฟังก์ชันสำหรับคำนวณหาค่า x^n เมื่อกำหนดให้ n >= 0 จะได้

รูปแบบของการเรียกตัวเอง คือ `pow(x,n) = x*pow(x,n-1)`

เงื่อนไขหยุด คือ `n=0` ซึ่งให้ค่า `pow(x,n) = pow(x,0) = 1`

ฟังก์ชันเรียกตัวเองที่สมบูรณ์แสดงได้ดังตัวอย่าง

```
#include <stdio.h>
double pow(int x, int n);
int main() {
      printf("Enter x and n:");
      scanf("%d %d",&x,&n);
      printf("Enter pow(x,n)=%0.2f",pow(x,n));
}
double pow(int x, int n) {
      if(n==0)                        //stop condition, when n==0
            return 1;                 //pow(x,0)=1
      else                            //case n>0
            return (x*pow(x,n-1)); //call recursive form
}
```

5. แบบฝึกหัด

1. จงเขียน recursive function เพื่อคำนวณหา Fibonacci number ลำดับที่ n เมื่อลำดับของ Fibonacci number เป็น 1 1 2 3 5 8 ...
2. เขียนโปรแกรมเพื่อคำนวณหา Fibonacci number โดยไม่ใช้ recursive function เพื่อเปรียบเทียบความแตกต่าง (ในแง่ความยากง่าย และความเร็วในการทำงาน)
3. ฝึกเขียนโปรแกรมเพื่อแก้ปัญหาตัวอย่างที่ 2-6 (ในหัวข้อความสัมพันธ์เวียนเกิด) โดยอาศัย recursive function
4. จงเขียนโปรแกรมเพื่อแก้ปัญหา tower of Hanoi (รายละเอียดอธิบายเพิ่มเติมในคาบ)
5. จงเขียนโปรแกรมเรียงลำดับข้อมูลด้วยวิธีการ merge sort โดยอาศัย recursive function
6. จงเขียนโปรแกรมเพื่อค้นหาข้อมูลในรายการข้อมูลที่เรียงลำดับแล้ว ด้วยวิธีการของ binary search โดยอาศัย recursive function